DLLs the Dynamic Way

MicHael Galkovsky
Pervasive Software

November 1999

Summary: Describes a macro class that simplifies and automates the dynamic linking of DLLs. (8 printed pages)

Contents

Introduction

Dynamic link libraries (DLLs) are a very powerful feature of the Microsoft® Windows® operating system. Like most developers, I have often found the need to link to various DLLs. When these DLLs are not Component Object Model (COM) libraries, linking to them has proven to be a big headache. The required steps are cumbersome and non-intuitive for object-oriented programmers like myself, and I have always ended up reusing old code. After several attempts to dig up old examples, I decided to write a macro class to simplify and automate the dynamic linking of DLLs. This class is abstract and allows a developer to extend the functionality if needed.

The Old Way

According to "About Dynamic Link Libraries" in the Platform SDK, there are two methods for calling DLLs:

- "In *load-time dynamic linking*, a module makes explicit calls to exported DLL functions. This requires you to link the module with the import library for the DLL. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded."
- "In *run-time dynamic linking*, a module uses the **LoadLibrary** or **LoadLibraryEx** function to load the DLL at run time. After the DLL is loaded, the module calls the **GetProcAddress** function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by **GetProcAddress**. This eliminates the need for an import library."

The two major benefits to using run-time linking are:

- When the DLL is not available, an application using load-time dynamic linking simply terminates, while the run-time dynamic linking is able to respond to the error.
- If the DLL changes, an application that uses load-time dynamic linking may again terminate, while an application linked at run-time is only affected if the desired functions are not present in the new DLL.

Run-time dynamic linking is not unlike querying interfaces in COM. It is more robust, however, and allows for more

flexibility on the part of DLL provider and consumer. This article assumes that you will want to link DLLs using run-time dynamic linking.

Traditional Linking—A Sample

Let's assume that you need to link to a DLL that keeps track of your art collection. This DLL, art.dll, has the following functions:

```
short FindArtist(LPCTSTR artist);
short AddArtist(LPCTSTR artist, LPCTSTR country, short century);
short FindArtifact(LPCTSTR name, LPCTSTR artist);
short AddArtifact(LPCTSTR artist, LPCTSTR name, short price);

// art.c
// A simple program that uses LoadLibrary and
// GetProcAddress to access FindArtist function from art.dll.
#include <stdio.h>
#include <windows.h>

//Define the function prototype
typedef short (CALLBACK* FindArtistType)(LPCTSTR);

void main(void)
{
   BOOL freeResult, runTimeLinkSuccess = FALSE;
   HINSTANCE dllHandle = NULL;
   FindArtistType FindArtistPtr = NULL;

   //Load the dll and keep the handle to it
   dllHandle = LoadLibrary("art.dll");

   // If the handle is valid, try to get the function address.
   if (NULL != dllHandle)
   {
      //Get pointer to our function using GetProcAddress:
      FindArtistPtr = (FindArtistType)GetProcAddress(dllHandle,
         "FindArtist");

      // If the function address is valid, call the function.
      if (runTimeLinkSuccess = (NULL != FindArtistPtr))
      {
         LPCTSTR myArtist = "Duchamp";
         short retVal = FindArtistPtr(myArtist);
      }

      //Free the library:
      freeResult = FreeLibrary(dllHandle);
   }

   //If unable to call the DLL function, use an alternative.
   if(!runTimeLinkSuccess)
```

```
        printf("message via alternative method\n");
}
```

This code sample was based in part on the code in the "Using Run-Time Dynamic Linking" section of the Platform SDK.

Traditional Linking—The Steps

These are the steps you need to follow every time you want to link to a DLL from your application:

1. Define the function prototypes.

```
typedef short (CALLBACK* FindArtistType)(LPCTSTR);
```

2. Load the DLL using the **LoadLibrary** function.
3. Keep the handle to the DLL instance.

```
dllHandle = LoadLibrary("art.dll");
```

4. Get a pointer to each function using the **GetProcAddress** function.
5. Cast function pointers to the types defined in Step 1.

```
FindArtistPtr = (FindArtistType)GetProcAddress(dllHandle,
      "FindArtist");
```

6. Verify function pointers.

```
if (runTimeLinkSuccess = (NULL != FindArtistPtr))
```

7. Use functions.

```
short retVal = FindArtistPtr(myArtist);
```

8. Unload the DLL.

```
    freeResult = FreeLibrary(dllHandle);
```

If you want to use art.dll in several applications, you will need to include this code in all of them. If you are going to use more then one function, then the workload increases. You will need to declare more types and check for more NULL pointers. Assigned function pointers are easily confused; and when you load multiple DLLs with multiple functions, it's even easier to lose track of what points to what. After doing this several times, I wrote the **PDLL** class.

PDLL Class

I wanted to write DLL loading code once and then be able to forget about it. I wanted to be able to include a header file in my project and start using the functions from the DLL. And I wanted to be able to do this with any DLL at any time. I originally set off to accomplish this with C++ templates but then quickly realized that, because types of parameters and names of the functions are not known up front, I had to use macros. After hours of fighting with macros and painful debugging, the solution started to take form.

Let's first look at the steps required for using the **PDLL** class:

1. Declare a class, and inherit it from the **PDLL** class.
2. Call the **DECLARE_CLASS** macro, and pass it the class name.
3. Call the **DECLARE_FUNCTION** macro for every needed function from the DLL.

The code looks like this:

```
//ArtDll.h
//header file for ArtDll class
#include "pdll.h"
class ArtDll: public PDLL
{
   DECLARE_CLASS(ArtDll)
//we call macros according to how many parameters the function
//takes, we pass to the macros the following parameters
//return type, function name, parameters
   DECLARE_FUNCTION1(short, FindArtist, LPCTSTR)
}
```

You are now ready to use the new class that encapsulates the DLL functionality.

```
//Art.cpp
#include "ArtDll.h"
//declare instance of the class
ArtDll myDll("art.dll");

//use the function from the dll
myDll.FindArtist(artist);
```

As you can see, the steps look a bit cleaner and are much more object-oriented.

Here is what the **PDLL** class can do for you:

- **Load the DLL**. The class lets you choose whether you want to specify the DLL name in class constructor or later by calling the **Initialize** function. If the requested DLL is not found, your application will not terminate. You can include your own custom messages informing the user about the missing DLL or provide alternative ways to handle the situation.
- **Unload the DLL**. Class inherited from **PDLL** keeps a static reference count of its instances. The DLL is unloaded after the last instance is destroyed, cutting down on resource usage within your application.
- **Obtain function pointers**. Functions are loaded when you use them, avoiding overhead at creation time. A function pointer is obtained using **GetProcAddress** the first time a call to the function is made. If **GetProcAddress** fails, your application will not crash.
- **Organize functions by DLL**. With the **PDLL** class, you can group your functions according to which DLL contains the functionality.

Under the Hood

Let's look under the hood of **PDLL** class in some detail. Most of the code is written using C macros. Macros are impossible to debug and are also very sensitive to spaces and alignments. If you plan to change these macros, be careful.

The backslash is used as a continuation character. Make sure that there are no blank spaces after the backslash. If you are new to macros, this may confuse you a bit. But the good news is that you don't have to worry about this if you don't plan to make modifications.

Function Declarations

Header file pdll.h contains macros that, when given a function description, create the typedef and declare a wrapper function. You'll need to call a distinct macro according to the number of arguments the desired function has. For a function with three parameters, you will need to call the macro **DEFINE_FUNCTION3**; for a function with two arguments, you will need to call the macro **DEFINE_FUNCTION2**, and so on. The wrapper functions that these macros declare contain code to make sure that the functions are loaded successfully and valid function pointers are obtained. These macros declare a short integer that indicates whether an attempt has been made to load the DLL functions. Function addresses are loaded the first time the functions are called, allowing you to keep the overhead to a minimum at load time. The function addresses can only be obtained if a valid DLL handle is obtained. If the DLL cannot be loaded or the attempt to obtain the function address has failed and a call is made to the function, the return value will be NULL cast to the return value of the function.

```
//macros for function declarations
//pdll.h
//function declarations according to the number of parameters
#define DECLARE_FUNCTION1(retVal, FuncName, Param1) \
   typedef  retVal (CALLBACK* TYPE_##FuncName)(Param1); \
   TYPE_##FuncName m_##FuncName; \
   short m_is##FuncName;\
   retVal FuncName(Param1 p1) \
```

```
    { \
       if (m_dllHandle) \
       { \
          if (FUNC_LOADED != m_is##FuncName) \
          {\
             m_##FuncName = NULL; \
             m_##FuncName = (TYPE_##FuncName)GetProcAddress(m_dllHandle,
#FuncName); \
             m_is##FuncName = FUNC_LOADED;\
          }\
          if (NULL != m_##FuncName) \
             return m_##FuncName(p1); \
          else \
             return (retVal)NULL; \
       } \
       else \
          return (retVal)NULL; \
    }
```

Class Declaration

You will need another macro to invoke the class declaration. It will ensure that you have a constructor that can initialize your DLL.

```
//declare constructors
// ClassName is the user class name
#define DECLARE_CLASS(ClassName) \
   public:   \
   ClassName (const LPCTSTR name){LoadDll(name);} \
   ClassName () {PDLL();}
```

Another Example

Let's walk through the whole process while creating a class for a compression DLL called compress.dll. Let's assume that this DLL has two functions that you want to use:

```
short CompressData(void* output, void* input, long size, short type)
short DecompressData(void* output, void* input, long size, short type)
```

Because you don't know whether or not the user will have this DLL on their system and you plan to update your compression DLL often (by adding new compression algorithms), **PDLL** class sounds like a good solution. You will need to declare a new class and inherit it from the **PDLL** class. This new class makes sure that it can load the DLL before the functions are used and allows you to update your compression DLL without recompiling the application.

You'll need to create a new header file that contains the definition for the wrapper class. Let's call this class **CompDll**. The header file should look like this:

```
//compress.h
//header file to contain the definition for CompDll class
#include "pdll.h"
class CompDll:public PDLL
{
//call the macro and pass your class name
DECLARE_CLASS(CompDll)
//use DECLARE_FUNCTION4 since this function has 4 parameters
DECLARE_FUNCTION4(short, CompressData, void*, void*, long, short)
DECLARE_FUNCTION4(short, DecompressData, void*, void*, long, short)
}
```

If you want the functions to show up in a Visual C++ drop-down function browser, you can declare a pure virtual class and then inherit the DLL class from it. The code looks like this:

```
//compress.h
#include "pdll.h"
//dll wrapper class using the virtual class
class VirtualComp
{
virtual short CompressData(void* input, void* output, long size, short
type) = 0;
virtual short DecompressData(void* input, void* output, long size,
short type) = 0;
}
class CompDll:public PDLL, VirtualComp
{
//call the macro and pass your class name
DECLARE_CLASS(CompDll)
//use DECLARE_FUNCTION4 since this function has 4 parameters
DECLARE_FUNCTION4(short, CompressData, void*, void*, long, short)
DECLARE_FUNCTION4(short, DecompressData, void*, void*, long, short)
}
```

When you want to use compress.dll, you will need to specify its name in the constructor of the class.

```
CompDll myDll("compress.dll");
```

Or you can hard code it by creating a constructor for the class.

```
class CompDll:public PDLL, public VirtualComp
{
//call the macro and pass your class name
DECLARE_CLASS(CompDll)
//use DECLARE_FUNCTION4 since this function has 4 parameters
DECLARE_FUNCTION4(short, CompressData, void*, void*, long, short)
DECLARE_FUNCTION4(short, DecompressData, void*, void*, long, short)
//hard code the dll name in the constructor
CompDll(){Initialize("compress.dll")};
}
```

MicHael Galkovsky lives in Austin, TX and is a software engineer on the Pervasive Software Developer Solutions team, where he is working on the next generation of database tools. Feel free to e-mail him at michael.galkovsky@pervasive.com.